The American University in Cairo

Computer Organisation and Assembly Language Programming

Fall 2024
Department of Computer Science and Engineering

Project 2 Report: Memory Hierarchy Simulator

Authors:

Name: Ahmed Ayman Elkhodary
ID: 900213472
Email: aae121@aucegypt.edu

Name: Ahmed Fawzy Abdelkader
ID: 900222633
Email: ahmed-fawzy@aucegypt.edu

Name: Omar Leithy
ID: 900221663
Email: omarleithym@aucegypt.eduu

Supervised by:
Dr. Cherif Salama

GitHub Repository: Ahmed-Fawzy14/Memory-Hierarchy-Simulator

December 7, 2024

Contents

1 Introduction

1.1 Project Overview

This project implements a cache simulator with a focus on direct-mapped caching for both instruction and data accesses. The simulator emulates a one-level read-only cache system for byte-addressable memory. A graphical user interface (GUI) was developed using Python's 'Tkinter' library, making the simulation process interactive and user-friendly.

The core objectives are:

- Simulate cache access patterns for instruction and data caches.
- Provide detailed metrics like hit ratio, miss ratio, and AMAT.
- Offer user-friendly interaction through a GUI for better accessibility.

The simulator is designed for educational purposes, demonstrating how caching systems impact memory access performance.

2 Implementation

2.1 Simulator Design

The cache simulator is divided into three main components:

- **Backend Logic**: Handles core simulation tasks, such as managing cache states, calculating indices and tags, and determining hits or misses.
- **Main Interface**: Serves as the entry point for running the simulator, managing input, and initiating the simulation.
- **GUI**: Built using 'Tkinter', it allows users to interact with the simulator through an intuitive interface.

2.2 Backend Functionality

The backend code implements a direct-mapped cache. Key operations include:

- **Cache Initialization**: The cache is initialized with configurable parameters, including the number of cache lines and access times.
- **Address Decoding**: Each memory address is divided into index, tag, and offset components.
- **Cache Access Handling**: For each memory access, the simulator checks if the address is present in the cache (hit) or absent (miss). Misses update the cache state by replacing the line's tag.
- **Statistics Calculation**: Tracks the number of accesses, hits, misses, and computes metrics such as hit ratio, miss ratio, and AMAT.

2.3 Main Code

The main script integrates the backend and GUI, orchestrating the overall simulation workflow:

- Handles user input for cache configuration.
- Provides the ability to switch between CLI and GUI modes.
- Generates simulation results in real-time, based on the input sequence.

2.4 GUI Features

The GUI simplifies the process of configuring and running the simulation. Key features include:

- **Configuration Panel**: Users can specify cache parameters such as line size, total size, and memory access times.
- **Load Address Sequence**: Allows users to load memory access patterns from a file or input them manually.
- **Simulation Controls**: Includes buttons for starting the simulation, stepping through memory accesses, and resetting the simulation.
- **Live Updates**: Displays cache statistics and visualization in real-time.

2.5 User Input and Output

The simulator requires the user to provide specific configuration parameters during initialization. These parameters are validated to ensure consistency:

- **Memory Access Time in Cycles**: The time required to access main memory, specified in cycles (e.g., 100).
- **Total Cache Size in Bytes**: Specifies the size of the cache (e.g., 64 bytes). Must be a power of 2.
- **Line Size in Bytes**: Specifies the size of each cache line (e.g., 16 bytes). Must be a power of 2 and smaller than the total cache size.
- **Cache Access Time in Cycles**: The time required to access the cache (e.g., 5 cycles).

The simulator performs input validation to ensure consistency:

- Total cache size must be divisible by the line size.
- Cache access time must be less than memory access time.
- All parameters must be positive integers.

Invalid input prompts an error message, and the user is asked to re-enter the values. This ensures the simulator is initialized correctly.

3 Bonus Features

The project includes two bonus features:

3.1 GUI Implementation

The GUI enhances user interaction and visualization. Built with 'Tkinter', it offers:

- File-based program loading for memory access sequences.
- Step-by-step execution of memory accesses with dynamic updates to the cache state.

In this project, we implemented separate caches for instructions and data. The user provides two distinct access sequences:

- **Instruction Access Sequence**: A file containing memory addresses accessed by the instruction cache.
- **Data Access Sequence**: A file containing memory addresses accessed by the data cache.

The simulator processes these access sequences independently for each cache, allowing separate tracking of hits, misses, and statistics for both instruction and data caches. This feature ensures that the simulation more accurately reflects the behavior of modern processors, which often have separate caches for instructions and data.

4 Testing

4.1 Test Programs

Several test cases were used to validate the simulator's functionality:

- **Sequential Access**: A series of sequential memory accesses to test cache misses and replacements.
- **Repeated Access**: Memory accesses targeting the same addresses to ensure cache hits occur after the initial misses.
- **Mixed Patterns**: Combines sequential and random accesses to emulate real-world scenarios.

5 Known Bugs and Limitations

5.1 Bugs

Throughout the development process, several bugs were encountered and resolved. Key issues included:

- **Index and Tag Calculation**: During initial implementation, the calculation logic for tag and index was incorrect, leading to invalid cache lookups. This was resolved by using the formula: `tag = address // (lineSize * numLines)` and `index = (address // lineSize) % numLines`.

- **Miss Handling**: Cache misses were not properly recorded because the cache line's valid bit and tag were not updated after a miss. This was fixed by setting `valid=1` and updating the tag field upon a miss.
- **AMAT Calculation**: Early versions used integer division in the AMAT formula, which resulted in zero values for certain cases. The formula was corrected to use float division.
- **Independent Cache Handling**: Initially, both instruction and data access sequences were processed by the same cache due to a reference error. This was fixed by ensuring each sequence passed its respective cache and statistics structures.
- **Large Address Handling**: For very large addresses close to the maximum representable by `addressBits`, indexing issues arose. Adjustments were made to handle large integers appropriately.
- **AMAT Bug**: A subtle bug in the AMAT formula was found and corrected. The correct formula is:

$$\text{AMAT} = \text{CacheAccessTime} + (\text{MissRatio} \times \text{MemoryAccessTime})$$

- **File Handling**: Errors were encountered when invalid or missing input files were loaded. Error handling was added to provide user-friendly error messages and prevent crashes.

5.2 Limitations

The simulator has the following limitations:

- Only supports direct-mapped caches; associative and multi-level caches are not implemented.
- Assumes a sparse memory model, which stores only accessed memory locations. This may not fully emulate dense address spaces used in real hardware systems.
- The GUI may experience lag when handling very large input sequences due to its synchronous update mechanism.

6 Design Decisions and Assumptions

6.1 Direct-Mapped Cache

A direct-mapped cache was chosen for its simplicity, making it suitable for educational purposes and easier to implement within the project constraints.

6.2 Sparse Memory Representation

Memory is represented sparsely to reduce overhead, storing only the memory locations accessed during the simulation.

7 User Guide

7.1 Development Environment

The simulator was developed using Python and tested in PyCharm. Required dependencies include ‘Tkinter‘.

7.2 Running the Simulator

To run the simulator:

1. Install Python and required libraries.
2. Open the project in PyCharm or your preferred IDE.
3. Run the ‘main.py‘ file to start the GUI or CLI-based simulator.

7.3 Loading Input Files

Input files should be plain text files with memory addresses separated by commas or newlines. These files can be loaded through the GUI.

7.4 Interactive GUI Usage

1. Configure cache parameters in the GUI.
2. Load memory access patterns via the ”Load Program” button.
3. Step through the simulation or execute it fully using the available controls.

8 Experience and Reflection

The project taught us valuable lessons in low-level programming, exposure to new programming languages like Python, CPU architecture, and user interface design, as part of the Computer Organization and Assembly Language Programming course. Challenges included handling instruction decoding, memory management, and GUI integration, which required teamwork and iterative debugging to solve.

A Screenshots of GUI and Simulation Outputs

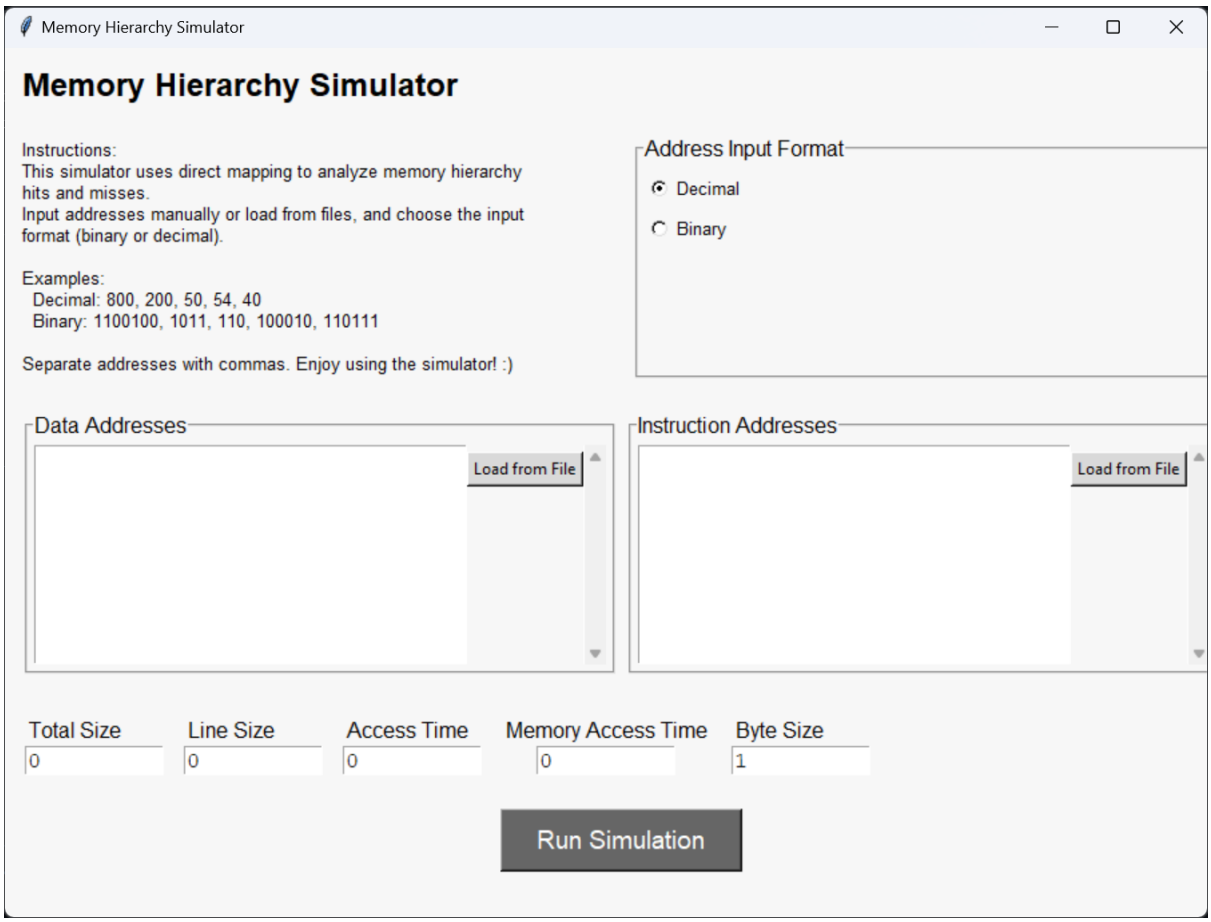
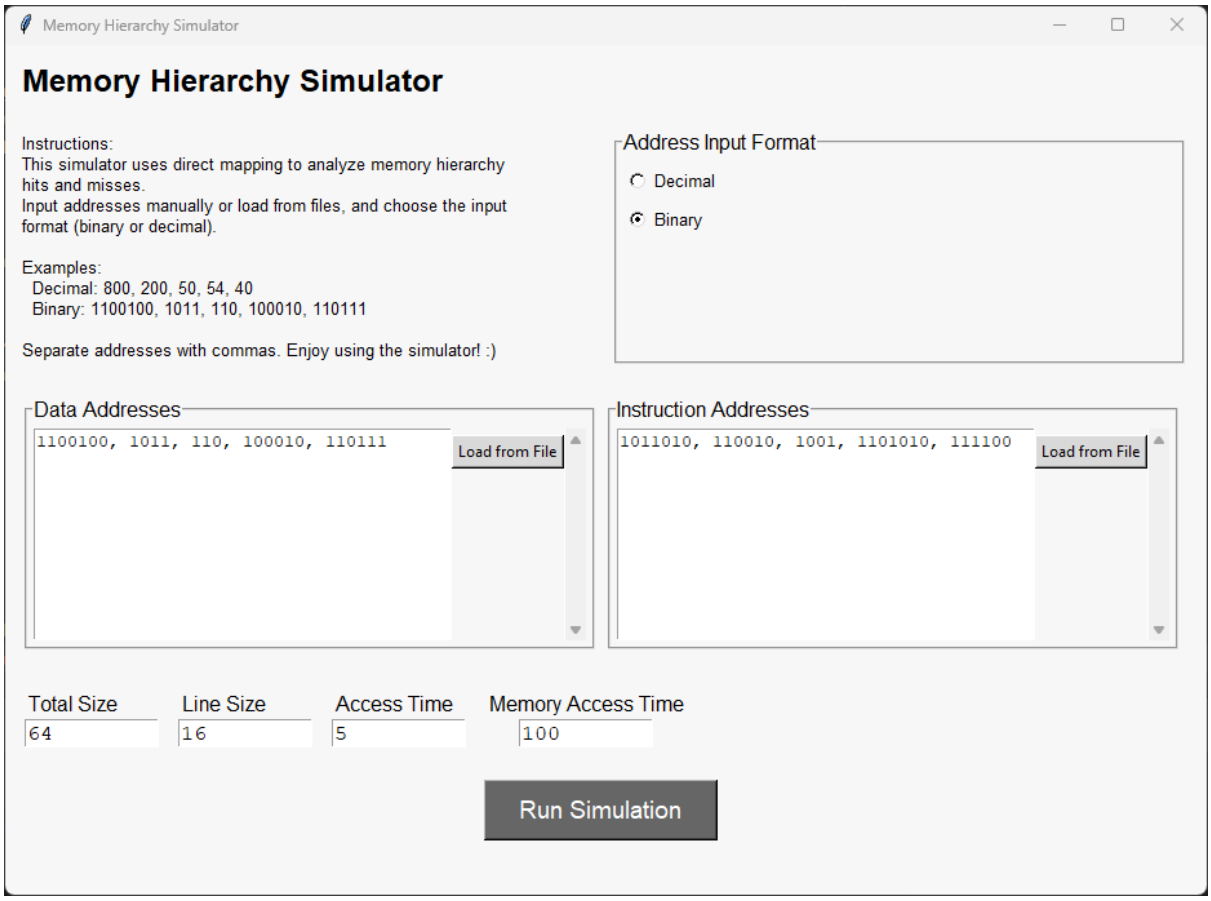


Figure 1: Simulation Window



Input your `Data Addresses` and `Instruction Addresses` manually or you can load a .txt file from your file directory. Specify your values for `Memory Access Time (in cycles)`, `Byte Size`, `Total Cache Size (in bytes)`, `Line Size (in bytes)`, `Cache Access Time (in cycles)` according to the range of values accepted. Then, select which address input format you want (Decimal/Binary) then click `Run Simulation`.

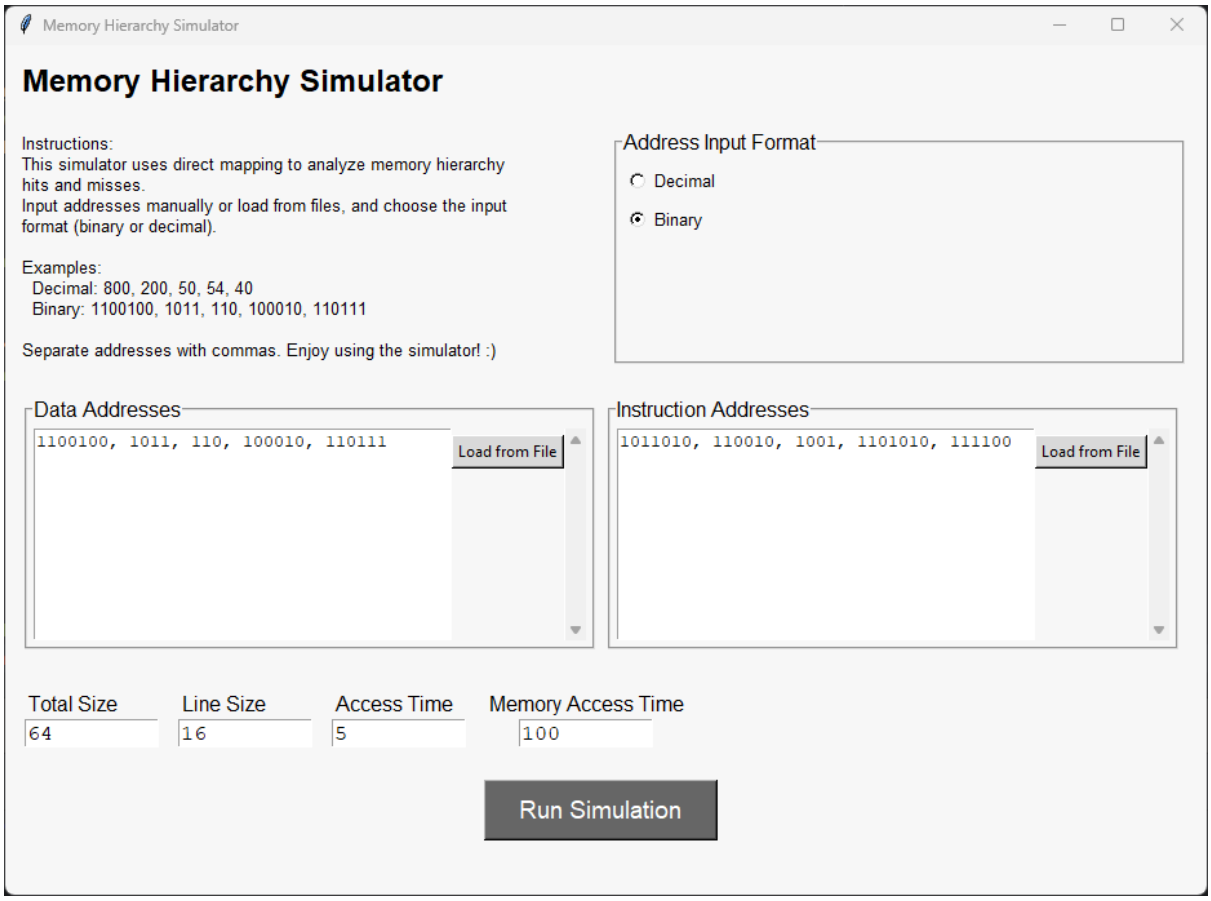


Figure 2: In Binary Format

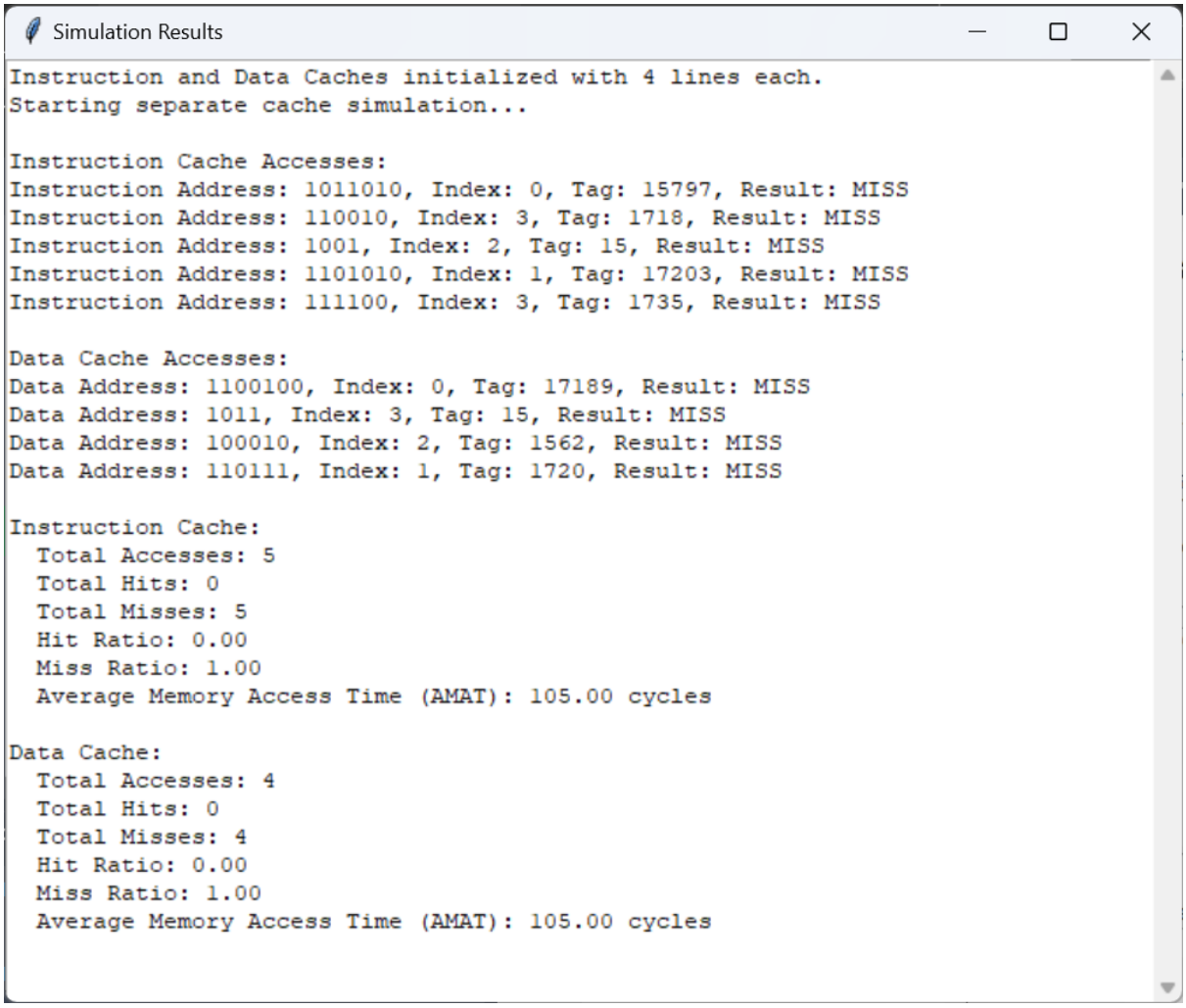


Figure 3: Simulation Result based on the previous inputted values. The output should be the same whether you choose Binary or Decimal format.